Introduzione agli Algoritmi

Pietro Di Lena

Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2021/2022

Introduzione

- Cos'è un algoritmo?
  - Procedura per risolvere un problema in un numero finito di passi
- Qual è l'origine della parola "algoritmo"?
  - Deriva dal nome del matematico Persiano Muhammad ibn Musa al-Khwarizmi
  - al-Khwarizmi latinizzato in Algoritmi
  - Autore di Algoritmi de numero Indorum (820), responsabile della diffusione del sistema numerico Hindu-Arabico in Medio Oriente ed Europa

Musa al-Khwarizmi (c. 780 – c. 850)

Algoritmo vs Programma

- Un algoritmo è una descrizione di *alto livello* di una procedura
  - Gli algoritmi esistono da prima dell'avvento dei computer
  - Non può essere eseguito su un computer
  - Può assumere un quantitativo illimitato di memoria
- Un programma è l'implementazione di un algoritmo
  - Deve essere scritto in qualche linguaggio di programmazione
  - Può essere eseguito su un computer
  - Deve tenere in conto i limiti di memoria

Prospettiva storica (incompleta)

- Euclide ($\sim$300 AC), Eratostene ($\sim$200 AC)
  - calcolo aritmetico, geometrico
- al-Khwarizmi ($\sim$800)
  - calcolo aritmetico, algebraico, geometrico
- Newton ($\sim$1700)
  - calcolo differenziale, integrale
- Turing ($\sim$1940)
  - crittoanalisi, risolvibilità

Ingredienti di un algoritmo

- Un algoritmo:
  - prende in input alcuni valori
  - esegue una sequenza finita di istruzioni
  - produce (eventualmente) un output
- Un algoritmo rappresenta la soluzione ad un problema da risolvere
  - La definizione del problema vincola input e output
  - Ci sono infiniti set di istruzioni che risolvono lo stesso problema
    - Esattamente lo stesso output per lo stesso input

Esempio: algoritmo di Euclide

Calcolare il Massimo Comune Divisore (MCD) tra x e y ($x \geq y$)

1. Dividi x per y e chiama r il resto della divisione
2. Se $r = 0$ allora y è il MCD tra x e y
3. Se $r \neq 0$ allora assegna y ad x, r ad y e ritorna al punto 1

```
function MCD(INT x, INT y) → INT
    if y == 0 then
        return x
    else
        r = x mod y
        return MCD(y,r)
```

- Algoritmo di Euclide in pseudo-codice: linguaggio umano sulla sinistra, linguaggio più vicino a quello di un calcolatore sulla destra
- Nota: assumiamo che $x \geq y$ ma possiamo rilassare tale assunzione modificando di poco il nostro pseudo-codice

Cosa ci serve per poter sviluppare algoritmi?

- Capire il problema che vogliamo risolvere
  - Quali sono i possibili input e output?
  - Come sono gli input mappati sugli output?
  - Ci sono proprietà matematiche legate al nostro problema?

- Apprendere come stimare l'efficienza di un algoritmo
  - Tempo: quanto è veloce un algoritmo?
  - Memoria: di quanta memoria ha bisogno?
  - Come possiamo individuare l'algoritmo migliore tra due scelte?

- Studiare tecniche algoritmiche e strutture dati note
  - Problemi differenti spesso condividono la stessa struttura di base
  - Molti problemi possono essere risolti modificando algoritmi noti
  - Come possiamo essere sicuri che la nostra soluzione sia corretta?

Problema algoritmico: i numeri di Fibonacci

- I numeri di Fibonacci sono una sequenza

  $$F_1,\ F_2,\ \ldots,\ F_n,\ \ldots$$

  in cui ogni numero è dato dalla somma dei due numeri precedenti

- I numeri di Fibonacci possono essere definiti da una relazione ricorsiva

$$F_n = \begin{cases} 1 & \text{se } n = 1 \text{ o } n = 2 \\ F_{n-1} + F_{n-2} & \text{altrimenti} \end{cases}$$

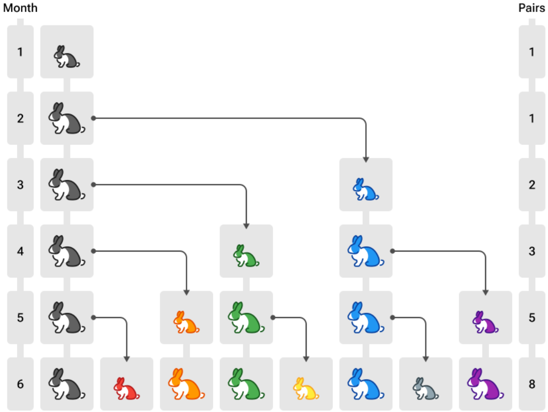
- Tipicamente si assume che $F_0 = 0$

Leonardo Fibonacci
(c. 1170 – c. 1250)

Cosa rappresenta la sequenza di Fibonacci?

Crescita (idealizzata) di una popolazione di conigli:

- Ogni coppia diventa fertile dopo un mese di vita
- A partire dal secondo mese la coppia dà alla luce una nuova coppia
- I conigli sono immortali



Algoritmo 1: formula chiusa

- Buona notizia: esiste una formula chiusa per calcolare il valore F_n

$$F_n = \frac{1}{\sqrt{5}} \left(\phi^n - \hat{\phi}^n \right)$$

  dove

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618 \text{ e } \hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

```
function FIB1(INT n) → INT
    return (1/√5) (φ^n − φ̂^n)        ▷ arrotondato ad int
```

- Cattiva notizia: approssimazioni numeriche per l'irrazionalità di ϕ e $\hat{\phi}$
  - Non è possibile calcolare esattamente F_n su un computer
  - Es. $\textsc{Fib1}(3) = 1.999863 \approx 2 = F_3$
  - Es. $\textsc{Fib1}(18) = 2583.023 \approx 2583 \neq F_{18} = 2584$
  - Aumentare la precisione di ϕ e $\hat{\phi}$ non aiuta
- Dobbiamo pensare ad un'altra soluzione

Algoritmo 2: soluzione ricorsiva "ingenua"

- Possiamo convertire direttamente la formula ricorsiva

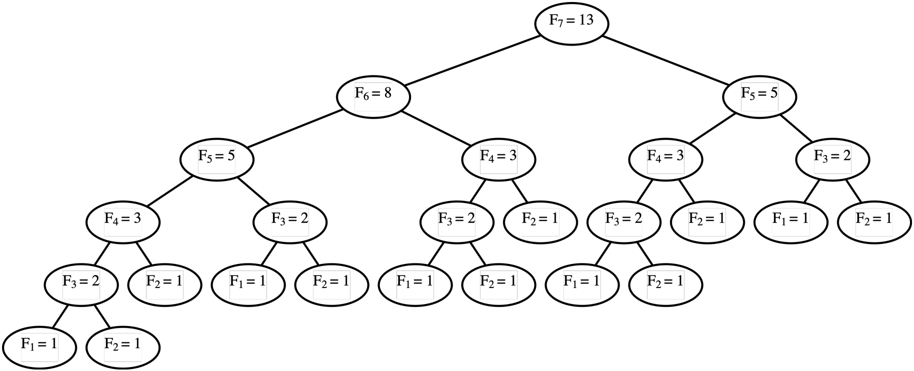
$$F_n = \begin{cases} 1 & n \leq 2 \\ F_{n-1} + F_{n-2} & n > 2 \end{cases}$$

in un algoritmo ricorsivo

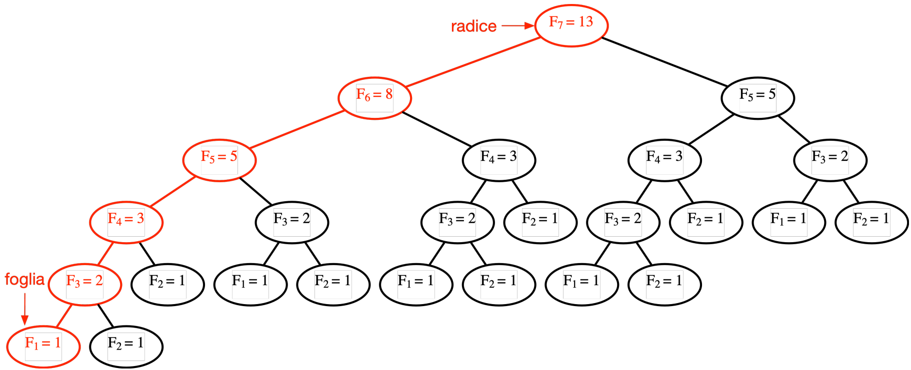
```
function Fib2(int n) → int
    if n ≤ 2 then
        return 1
    else
        return Fib2(n − 1)+ Fib2(n − 2)
```

- Quanta memoria richiede Fib2?
- Quanto è veloce Fib2?

Algoritmo 2: albero di ricorsione di Fibonacci



Algoritmo 2: stima della memoria

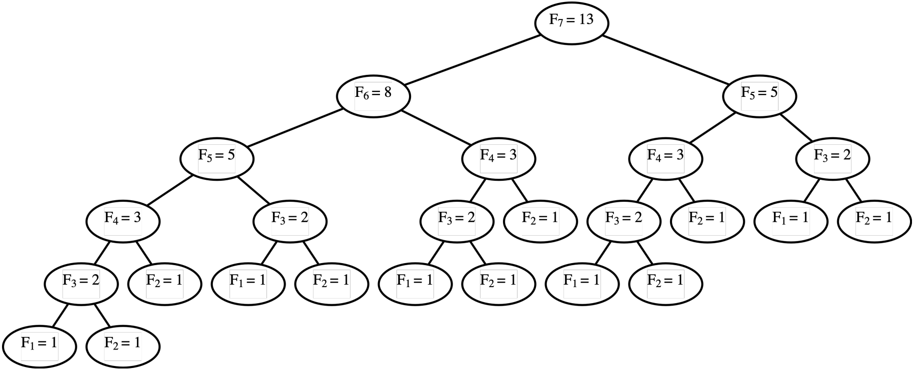


- Apparentemente, Fib2 richiede memoria solo per memorizzare n
- In realtà, le chiamate ricorsive sono gestite dal *call stack*
  - la memoria al tempo t dipende dalle chiamate attive al tempo t
- La memoria usata è *proporzionale* ad n nel caso peggiore
  - Il numero massimo possible di chiamate attive in Fib2 è limitato dal percorso *radice-foglia* più lungo nell'albero di ricorsione

Algoritmo 2: stima del tempo di calcolo

- Dovremmo misurare il tempo di calcolo in secondi?
  - Dipende dalla macchina su cui eseguiamo il programma
- Dovremmo contare il numero di istruzioni-macchina nell'algoritmo?
  - Stesso problema: dipende dal linguaggio di programmazione
  - Difficile stimare il numero di istruzioni-macchina da pseudocodice
- Approccio indipendente dalla macchina e dal linguaggio:
  - contiamo il numero di **operazioni elementari** nello pseudocodice

Algoritmo 2: stima del tempo di calcolo



- Idea: stimiamo il numero di chiamate ricorsive

$$T(n) = \text{numero di nodi nell'albero di ricorsione di } F_n$$

- Ogni chiamata ricorsiva richiede alla peggio un numero constante di operazioni elementari: una somma, due chiamate ricorsive, istruzione di return (stesso costo per ogni chiamata)

Algoritmo 2: stima del tempo di calcolo

```
function Fib2(int n) → int
    if n ≤ 2 then
        return 1
    else
        return Fib2(n − 1)+ Fib2(n − 2)
```

- Domanda: come trovare un limite superiore/inferiore per $T(n)$?
- Risposta: estraiamo una relazione di ricorrenza dallo pseudocodice

$$T(n) = \begin{cases} 1 & n \leq 2 \\ T(n-1) + T(n-2) + 1 & n > 2 \end{cases}$$

L'albero $T(n)$ contiene 1 nodo + i nodi dei sotto-alberi $T(n\text{-}1), T(n\text{-}2)$

Algoritmo 2: limite inferiore per $T(n)$

$$T(n) = T(n-1) + T(n-2) + 1$$

$$\geq 2T(n-2) + 1$$ Poiché $T(n-1) \geq T(n-2)$

$$\geq 4T(n-4) + 2 + 1$$ Poiché $T(n-2) \geq 2T(n-4)+1$

$$\geq 8T(n-6) + 4 + 2 + 1$$ Poiché $T(n-4) \geq 2T(n-6)+1$

$$\geq \ldots$$

$$\geq 2^k T(n-2k) + \sum_{i=0}^{k-1} 2^i$$ Generalizzazione al k-esimo passo

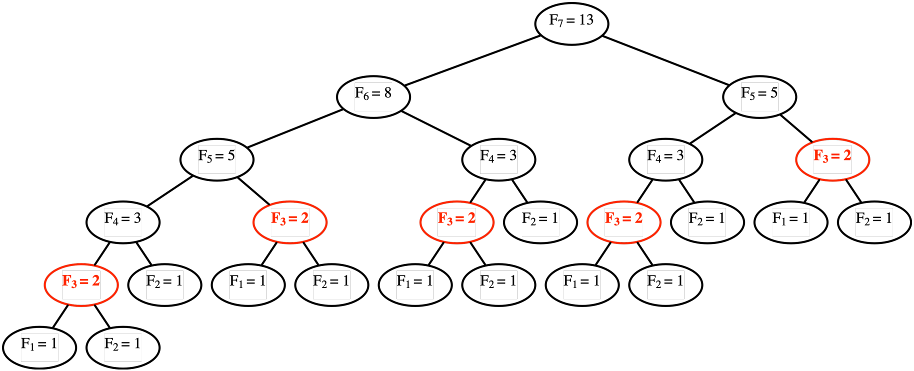
$$\geq \ldots$$ Stop quando $n-2k = 0 \Rightarrow k = \lfloor n/2 \rfloor$

$$\geq 2^{\lfloor n/2 \rfloor} + \frac{2^{\lfloor n/2 \rfloor} - 1}{2-1}$$

$$\geq 2^{\lfloor n/2 \rfloor}$$

Soluzione: $T(n)$ ha un numero esponenziale di nodi maggiore di $2^{n/2}$

Algoritmo 2: qual è il problema?



- La maggior parte dei numeri sono ri-calcolati più di una volta
  - La ricorsione non ricorda i numeri precedentemente calcolati
- Possiamo fare di meglio?

Algoritmo 3: soluzione iterativa

- La ricorsione non è una buona idea, cerchiamo una soluzione iterativa

- Idea: usiamo un array di lunghezza n per memorizzare $F_1, F_2, ... F_n$

```
function Fib3(int n) → int
    let F[1, . . . , n] be a new array of int
    F[1] = 1
    F[2] = 1
    for i = 3, . . . , n do
        F[i] = F[i − 1]+F[i − 2]
    return F[n]
```

- Quanto è veloce Fib3?
- Quanta memoria richiede Fib3?

Algoritmo 3: stima del tempo e memoria

- Cerchiamo di stimare
  - tempo di calcolo: contiamo il numero di operazioni elementari
  - memoria usata: sommiamo la dimensione delle variabili usate

```
function Fib3(int n) → int
    let F[1, ..., n] be a new array of int      ▷ 1 volta (creazione)
    F[1] = 1                                    ▷ 1 volta (assegnamento)
    F[2] = 1                                    ▷ 1 volta (assegnamento)
    for i = 3, ..., n do                        ▷ n-2 volte (assegnamento)
        F[i] = F[i − 1]+F[i − 2]                ▷ n-2 volte (assegnamento+somma)
    return F[n]                                 ▷ 1 volta (return)
```

- Tempo di calcolo *proporzionale* ad n
  - Totale: $4 + 3(n - 2) = 3n - 2$ operazioni elementari (per $n > 2$)
- Memoria *proporzionale* ad n (array F e variabili i, n)
  - Abbiamo davvero bisogno di utilizzare un array di lunghezza n?

ALGORITMO 4: SOLUZIONE EFFICIENTE IN MEMORIA

- Idea: per calcolare F_n possiamo memorizzare unicamente F_{n-1} eF_{n-2}

```
function FIB4(INT n) → INT
    a = 1
    b = 1
    for i = 3, ..., n do
        c = a + b
        a = b
        b = c
    return b
```

- Quanto è veloce FIB4?
- Quanta memoria richiede FIB4?

Algoritmo 4: stima del tempo e della memoria

- Come prima, contiamo il numero di operazioni elementari e variabili

```
function Fib4(int n) → int
    a = 1                          ▷ 1 volta (assegnamento)
    b = 1                          ▷ 1 volta (assegnamento)
    for i = 3, . . . , n do        ▷ n-2 volte (assegnamento)
        c = a + b                  ▷ n-2 volte (assegnamento+somma)
        a = b                      ▷ n-2 volte (assegnamento)
        b = c                      ▷ n-2 volte (assegnamento)
    return b                       ▷ 1 volta (return)
```

- Tempo di calcolo proporzionale ad n
  - Totale: $3 + 5(n - 2) = 5n - 7$ operazioni (più lento di Fib3?)
- Memoria costante
  - Usiamo cinque variabili (n, i, a, b, c), indipendentemente da n

Possiamo ritenerci soddisfatti?

Teorema

Consideriamo $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Per ogni $n \geq 2$ abbiamo che

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$$

Dimostrazione (per induzione su n)

1. Caso base. Per $n = 2$, $A^1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$ è vera
2. Caso induttivo. Assumiamo che per qualche $k \geq 2$ sia vero che
$$A^{k-1} = \begin{pmatrix} F_k & F_{k-1} \\ F_{k-1} & F_{k-2} \end{pmatrix}$$
allora dobbiamo mostrare che la proprietà è vera per $A^k = A^{k-1} \times A$

$$A^k = \begin{pmatrix} F_k & F_{k-1} \\ F_{k-1} & F_{k-2} \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_k + F_{k-1} & F_k \\ F_{k-1} + F_{k-2} & F_{k-1} \end{pmatrix} = \begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix}$$

Induzione matematica

- L'induzione matematica è una tecnica di dimostrazione matematica
- Consideriamo una proprietà P sui numeri naturali $\mathbb{N}$
- L'induzione matematica dice che P vale per tutti i numeri naturali se
  1. Caso base: P è vera per il valore iniziale (es. 0)
  2. Caso induttivo: assumendo che P sia vera per n possiamo dimostrare che P è vera per $n + 1$
- Non dobbiamo necessariamente considerare tutti i numeri naturali
  - Esempio: la proprietà P è vera per tutti i valori $n \geq 2$
- Possiamo avere più di una assunzione nel caso induttivo
  - Esempio: assumendo che P sia vera per n ed $n + 1$ dimostriamo che P è vera per $n + 2$
  - In tal caso, bisogna mostrare anche che P è vera per 2 casi base

Algoritmo 5: soluzione con matrici

- Idea: usiamo il Teorema precedente

```
function Fib5(int n) → int
    A = ( 1 1 ; 1 0 )
    for i = 2, ..., n do
        A = A × ( 1 1 ; 1 0 )        ▷ × indica il prodotto di matrici
    return A[1][1]
```

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$A = A \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

- Tempo di calcolo proporzionale ad n
  - $n-1$ moltiplicazioni di matrici $+\ 2n-1$ assegnamenti $+\ 1$ return
- Memoria costante
  - Usiamo 3 variabili (A, i, n), indipendentemente da n
- Nessun miglioramento rispetto a Fib4 ma ...

Potenza di matrici velocizzata

- Possiamo velocizzare il calcolo della potenza di matrici
- Idea: se n è pari allora $A^n = (A^{n/2})^2$, $A^{n+1} = (A^{n/2})^2 \cdot A$

function FibMatPow(FibMat M, int n) $\rightarrow$ FibMat
    if $n > 1$ **then**
        $A =$ FibMatPow$(M, n/2)$
        if $n \mod 2 == 0$ **then** ▷ *n pari*
            $M = A \times A$
        else ▷ *n dispari*
            $M = A \times A \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
    return M

- Tempo di calcolo proporzionale a $\log_2 n$
  - Per calcolare A^n eseguiamo $\approx \log_2 n$ chiamate ricorsive ($2^i = n$)
- Memoria proporzionale a $\log_2 n$
  - Memoria costante (A, M, n) per ognuna delle $\log n$ chiamate

ALGORITMO 6: SOLUZIONE CON POTENZA VELOCE

- Idea: usiamo l'algoritmo velocizzato per la potenza di matrici

```
function Fib6(int n) → int
    M = ( 1 1 ; 1 0 )
    M = FibMatPow(M, n − 1)
    return M[1][1]
```

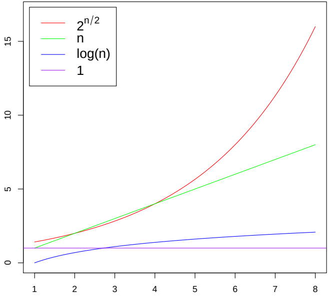
$$
\begin{array}{l}
\textbf{function } \textsc{Fib6}(\textsc{int } n) \rightarrow \textsc{int} \\
\quad M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\
\quad M = \textsc{FibMatPow}(M, n-1) \\
\quad \textbf{return } M[1][1]
\end{array}
$$

- Tempo di calcolo proporzionale a $\log_2 n$
  - Equivalente al tempo di calcolo di FIBMATPOW
- Memoria proporzionale a $\log_2 n$
  - Occupazione di memoria di FIBMATPOW

Sommario

- Tempo di calcolo e occupazione di memoria in notazione asintotica

Algoritmo	Tempo	Memoria
Fib2	$\Omega(2^{n/2})$	$O(n)$
Fib3	$O(n)$	$O(n)$
Fib4	$O(n)$	$O(1)$
Fib5	$O(n)$	$O(1)$
Fib6	$O(\log n)$	$O(\log n)$



- Siamo partiti da un algoritmo con tempo esponenziale di calcolo (inutilizzabile) e siamo arrivati ad un algoritmo logaritmico (efficiente)
- Benché tutti gli algoritmi sono equivalenti in termini di risultato, la loro efficienza computazionale li rende utilizzabili o meno
- Vedremo meglio la notazione asintotica, per il momento diciamo che ..

Valore in input vs Dimensione dell'input

- Abbiamo stimato l'efficienza rispetto al valore in input n
- Normalmente valutiamo l'efficienza in termini di dimensione dell'input
- La dimensione dell'input è il numero di bit sufficienti per rappresentarlo

$$|n| = \lceil \log_2 n \rceil \approx \log_2 n \quad \Rightarrow n = 2^{|n|}$$

- Tempi di calcolo e memoria in termini della dimensione dell'input

Algorithm	Time	Space
Fib2	$\Omega(2^{2^{\|n\|}})$	$O(2^{\|n\|})$
Fib3	$O(2^{\|n\|})$	$O(2^{\|n\|})$
Fib4	$O(2^{\|n\|})$	$O(1)$
Fib5	$O(2^{\|n\|})$	$O(1)$
Fib6	$O(\|n\|)$	$O(\|n\|)$

- Nota. Il tempo di calcolo (in sec) non cambia, cambia solo come valutiamo in tempo di calcolo di un algoritmo: in termini della dimensione dell'input e non del valore in input